CS 330 – Spring 2023 – Homework 06

Due: Wednesday 4/5 at 11:59 pm on Gradescope

Collaboration policy Collaboration on homework problems is permitted, you are allowed to discuss each problem with at most 3 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (these include anyone not enrolled in the class) is strictly forbidden.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

Typesetting Solutions should be typed and submitted as a PDF file on Gradescope. You may use any program you like to type your solutions. LaTeX, or "Latex", is commonly used for technical writing (`overleaf.com` is a free web-based platform for writing in Latex) since it handles math very well. Word, Google Docs, Markdown or other software are also fine.

Solution guidelines For problems that require you to provide an algorithm, you must give the following:

1. a precise description of the algorithm in English and pseudocode (*),
2. a proof of correctness,
3. an analysis of the asymptotic running time of your algorithm.

You may use anything we learned in class without further explanation. This includes using algorithms from class as subroutines, stating facts that we proved in class, e.g. correctness of subroutines, running time of subroutines and use the notation. Your description should be at the level that it is clear to a classmate who is taking the course with you.

You should be as clear and concise as possible in your write-up of solutions.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand.

(*) It is fine if the English description concentrates on the high level ideas and doesn't include all the details. But the reader should not have to figure out your solution solely based on the pseudocode. You can also add comments to your pseudocode, in fact that is best practice.

- By: Gianna Sfrisi U77992006
- Collaborators: Albert Slepak U00956163

Problem 1. Consider the following algorithm A:

```
1: procedure A(integer: n)
2:     if n = 0 then
3:         return 0
4:     if n is even then
5:         return A(n/2)
6:     else
7:         return A(2(n − 1)) + 1
```

(a) Prove that A terminates when its input n is a non-negative integer

Proof through Strong Induction: To prove P(K) we must prove P(K+1). If K+1 is even, then the next call is A((K+1)/2) since the algorithm divides all even values by 2. If K+1 is odd, then the next recursive call is A(2(K+1-1)) + 1 which simplifies to A(2K) + 1. Since any number multiplied by 2 is an even number, and P(K+1) = P(2K), the next call would divide P(2K) by 2. Hence P(K+1) = P(K) for any non-negative value of K.

(b) Write a recurrence for the running time (in terms of n) of A.

T(n/2) + $O(1)$ /* for even values of n */

T((n-1)/2) + $O(1)$ + $O(1)$ + $O(1)$ /* for odd values of n */

T($\lfloor n/2 \rfloor$)+O(1)/ $* Overall recurrence for the running time * /$

(c) Give a closed form for its asymptotic running time (using whichever method you like).

$O(log(n))$

(d) If we look at the tree of recursive calls to A made by this algorithm, what is its depth (asymptotically, in terms of n)? How many leaves does it have (asymptotically, in terms of n)?

The depth of the algorithm is $\log\_2 n$.

There is only 1 leaf node.

(e) What function of n does A compute? It's easy to express it in terms of the binary expansion of n (i.e. the bits you get when you write n in base 2).
It counts the number of bits that are set to 1 in the binary expansion of n.

Problem 2. Let T be a binary tree[1] with n vertices. Deleting any vertex v splits T into at most three subtrees, containing the left child of v (if any), the right child of v (if any), and the parent of v (if any). We call v a *central* vertex if each of these smaller trees has at most $n/2$ vertices. See Figure 1 for an example. Describe and analyze an algorithm to find a central vertex in an arbitrary given binary tree.

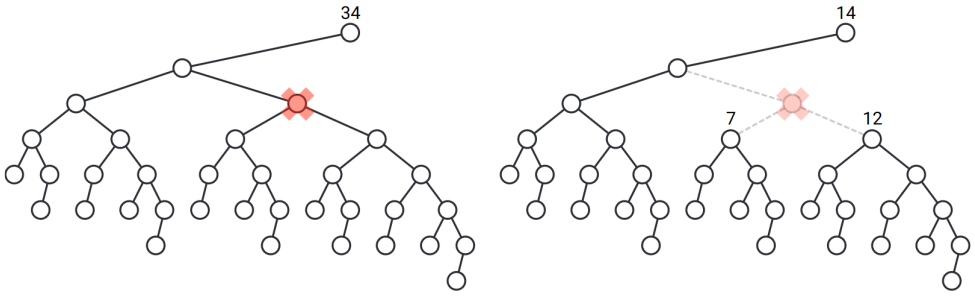

Figure 1: Deleting a central vertex in a 34-node binary tree, leaving subtrees with 14, 7, and 12 nodes.

Algorithm:

```
function assignSubtreeWeights(node) {
    if (node == null) {
        return 0;
    }
    node.leftSubtreeWeight = assignSubtreeWeights(node.leftChild);
    node.rightSubtreeWeight = assignSubtreeWeights(node.rightChild);

    nodeTotalWeight = node.leftSubtreeWeight + node.rightSubtreeWeight;
    return nodeTotalWeight + 1;
}
function findCentralVertex(node, parentSubtreeWeight) {
    if (node.leftSubtreeWeight ≤ maxAllowedNodes and node.rightSubtreeWight ≤ maxAllowedNodes
and parentSubtreeWeight ≤ maxAllowedNodes) {
        return node;
    }
    result = findCentralVertex(node.leftChild, parentSubtreeWeight - node.leftSubtreeWeight);
    if (result != null)
        return result;
    result = findCentralVertex(node.rightChild, parentSubtreeWeight - node.rightSubtreeWeight);
    if (result != null)
        return result;
    return null;
}
```

[1] In a binary tree, each node has between 0 and 2 children.

```
// Get total node count
nodeCount = assignSubtreeWeights(rootNode);
maxAllowedNodes = nodeCount / 2; // this rounds down

centralVertex = findCentralVertex(rootNode, nodeCount - 1);
```

Run-time Analysis:
The algorithm is broken up into two different helper functions. The first function assignSubtreeWeights traverses down the tree and assigns a weight to every node; hence it goes through each node one time and has a run-time of $O(n)$. The second function, findCentralVertex searches through the tree starting at the root node and finds a node where the weight of all the possible sub-trees has a weight less than n/2. In the worst possible case, the function will have to go through every node in the tree until it is found. Therefore, in the worst-case scenario, the function would have a run-time of $O(n)$ if every node has to be searched in order to find the central vertex. Hence the total run-time would be $O(2n)$ which simplifies to just $O(n)$

Proof of Accuracy:
The algorithm uses two recursive calls; one finds the weight of each node, and the second uses that weight to find the central vertex. The weight of each node is the total amount of nodes in each sub-tree as if the current node was the root. The first function, assignSubtreeWeights takes in the root node, uses recursion, and traverses down the tree. The function first goes down the left sub-tree and then traverses down the right sub-tree. Once the weight is assigned for each node the second recursive function, findCentralVertex is called, taking in the root node and the weight of the root node as parameters. The weight of the root node is calculated by the total amount of nodes minus 1 to take into account the root node itself. The maxAllowedNodes is the total amount of nodes in the tree divided by 2, given by the constraints of the problem. The first call of findCentralVertex tests if the weight of the left sub-tree, right sub-tree, and parent sub-tree are all less than the maxAllowedNodes; if all three possible sub-trees are, then the central vertex has been found. Additionally, if there are only two of the three possible sub-trees present, the weight of that sub-tree will be 0 which is automatically less than the maxAllowedNodes. If one of those trees has a weight greater than the maxAllowedNodes, then that node is not the central vertex and the function recursively calls the root of the left sub-tree. The function continues this pattern and if the central vertex is still not found, then the function traverses down the right sub-tree until the central vertex is found.